

Programación Orientada a Objetos

Lambdas y streams

Interfaces funcionales, referencias a métodos y la Stream API

Gerson Garrido

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Del comparador como clase a la lambda
2. Interfaces funcionales y `java.util.function`
3. Referencias a métodos y variables capturadas
4. Comparadores compuestos; colecciones que reciben lambdas
5. Streams: fuente, intermedias, terminal; evaluación lazy
6. `Collectors`, `groupingBy` y `Optional`

> **Lectura**
>
> Eck (v. 9): sec. 4.5, subsecs. 5.8.3–5.8.4, 10.4.3 y sec. 10.6.

SECCIÓN 01

Sesión A: lambdas

01

La semana pasada: un comparador es una clase

ByDuration.java

```java
public class ByDuration implements Comparator<
    Song> {

    @Override
    public int compare(Song a, Song b) {
        return Integer.compare(a.seconds(), b.
    seconds());
    }
}
```

uso

```java
songs.sort(new ByDuration());
```

- Seis líneas y un archivo para decir una idea: “por duración”.
- Diez órdenes distintos = diez archivos.
- ¿Qué parte es información de verdad y qué parte es ceremonia?

Paso 1: una clase anónima

ThreeWaysDemo.java

```java
Comparator<Song> byAnonymous = new Comparator<Song>() {
    @Override
    public int compare(Song a, Song b) {
        return Integer.compare(a.seconds(), b.seconds());
    }
};
```

- Una clase sin nombre, declarada e instanciada donde se usa. Ya no hay otro archivo.
- Pero: `Comparator<Song>` ya lo dice la variable; `compare` es el único método abstracto; los tipos de `a` y `b` salen de `Comparator<Song>`.
- Lo único que el compilador no puede adivinar: los parámetros y el cuerpo.

Paso 2: una lambda

ThreeWaysDemo.java

```
1  Comparator<Song> byLambda = (a, b) -> Integer.compare(a.seconds(), b.seconds());
2
3  Comparator<Song> byKey = Comparator.comparingInt(Song::seconds);
```

- Parámetros, flecha `->`, cuerpo. Nada más.
- `comparingInt` recibe la función que extrae la clave; `Song::seconds` es una referencia a método.

Una lambda es un objeto

Una forma corta de escribir `new Comparator<Song>() { ... }`. Se guarda en variables, se pasa como argumento, se devuelve.

Cuatro implementaciones, el mismo criterio de orden

1 · CLASE CON NOMBRE (SEMANA 11)

```
class ByDuration implements Comparator<Song> {
  public int compare(Song a, Song b) {
    return Integer.compare(a.seconds(), b.seconds()); } }
```

2 · CLASE ANÓNIMA

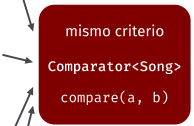
```
new Comparator<Song>() {
  public int compare(Song a, Song b) {
    return Integer.compare(a.seconds(), b.seconds()); } }
```

3 · LAMBDA

```
(a, b) -> Integer.compare(a.seconds(), b.seconds())
```

4 · STATIC FACTORY METHOD + REFERENCIA A MÉTODO

```
Comparator.comparingInt(Song::seconds)
```



en color, lo esencial; en gris, la ceremonia que la lambda ya no pide

Las cuatro ordenan igual

```bash
$ java -cp build ThreeWaysDemo
clase:        Arena  Malecón*  Cumbre*  Brisa  Garúa  Puente de piedra
anónima:      Arena  Malecón*  Cumbre*  Brisa  Garúa  Puente de piedra
lambda:       Arena  Malecón*  Cumbre*  Brisa  Garúa  Puente de piedra
comparingInt: Arena  Malecón*  Cumbre*  Brisa  Garúa  Puente de piedra
```

`ThreeWaysTest` lo fija: la lambda y `comparingInt` dan la misma lista que `new ByDuration()`. `*`: Malecón y Cumbre duran 190 s; `sort` es estable y no las reordena.

La sintaxis

Forma	Ejemplo
Un parámetro	`s -> s.seconds() > 200`
Varios parámetros	`(a, b) -> a + b`
Sin parámetros	`() -> new ArrayList<>()`
Tipos explícitos	`(Song a, Song b) -> ...`
Cuerpo en bloque	`s -> { int m = s.seconds() / 60; return m + " min"; }`

- Expresión sola: su valor es lo que devuelve, sin `return`.
- ¿Qué interfaz implementa? La decide el target type: la variable o el parámetro que la recibe.
- `var f = (String s) -> s.length();` no compila: no hay target type.

Ahora sí: la flecha de `assertThrows`

desde la semana 3

```
1  assertThrows(NoSuchElementException.class,
2          () -> s.pop());
```

LibraryReportsTest.java

```
1  Executable getFromEmpty =
2      () -> LibraryReports.longest(List.of()).
       orElseThrow();
3  assertThrows(NoSuchElementException.class,
       getFromEmpty);
```

- `Executable`: interfaz funcional de JUnit, `void execute()`.
- `assertThrows` llama a `execute()` dentro de un `try`.
- Sin la flecha, `s.pop()` se ejecutaría antes de llamar a `assertThrows`, y nadie capturaría la excepción.

Por dentro: `invokedynamic`

bash

```
$ ls build | grep ThreeWaysDemo
ThreeWaysDemo$1.class
ThreeWaysDemo.class
$ javap -c -p -cp build ThreeWaysDemo | grep invokedynamic
  16: invokedynamic ... // InvokeDynamic #0:compare:()Ljava/util/Comparator;
  22: invokedynamic ... // InvokeDynamic #1:applyAsInt:()...ToIntFunction;
```

- La clase anónima generó `ThreeWaysDemo$1.class`. Las lambdas, ningún archivo.
- El cuerpo queda como método privado; `invokedynamic` entrega un objeto que implementa `Comparator`.
- Si es nuevo o reutilizado lo decide la JVM: no comparen lambdas con `==`.

Salida recortada (`...`).

Interfaces funcionales

FunctionsDemo.java

```java
@FunctionalInterface
interface Discount {
    int apply(int cents);
}
Discount tenPercent = cents -> cents
      - cents / 10;
```

- Un solo método abstracto (más `default` y `static`).
- Solo ellas se implementan con lambdas.
- `@FunctionalInterface`: opcional; el compilador lo verifica.

javac · con un segundo método abstracto

```
error: Unexpected @FunctionalInterface annotation
  Discount is not a functional interface
    multiple non-overriding abstract methods found in interface Discount
```

java.util.function

Interfaz	Método	Recibe → devuelve	Para qué
`Predicate<T>`	`test`	`T` → `boolean`	filtrar
`Function<T, R>`	`apply`	`T` → `R`	transformar
`Consumer<T>`	`accept`	`T` → nada	imprimir, guardar
`Supplier<T>`	`get`	nada → `T`	fabricar
`UnaryOperator<T>`	`apply`	`T` → `T`	transformar sin cambiar tipo
`BinaryOperator<T>`	`apply`	`T`, `T` → `T`	combinar

Versiones para primitivos, sin boxing: `IntPredicate`, `ToIntFunction<T>`, `IntBinaryOperator`…

Funciones como objetos: se combinan

FunctionsDemo.java · resumido; el resultado de cada línea, en el comentario

```java
Predicate<Song> isLong = s -> s.seconds() > 200;
Function<Song, String> label = s -> s.artist() + " - " + s.title();
Function<String, Integer> length = String::length;
Song garua = Songs.sample().get(0);

isLong.test(garua)                    // true
isLong.negate().test(garua)           // false
label.apply(garua)                    // "Costa Verde - Garúa"
label.andThen(length).apply(garua)    // 19
```

- `negate`, `andThen`: métodos `default` que devuelven una función nueva.
- `andThen`: primero arma la etiqueta, después mide su largo (19 caracteres).

Referencias a métodos

Tipo	Referencia	Lambda equivalente
Estático	`Integer::sum`	`(a, b) -> Integer.sum(a, b)`
Estático	`IO::println`	`x -> IO.println(x)`
Instancia, objeto = 1.^{er} parámetro	`Song::title`	`s -> s.title()`
Instancia de un objeto concreto	`"Duna"::equals`	`x -> "Duna".equals(x)`
Constructor	`ArrayList::new`	`() -> new ArrayList<>()`

`Song::title` no nombra una canción: es una función que recibe una canción y le pide su título.
Si la lambda hace algo más que pasar parámetros, se queda como lambda.

Variables capturadas: effectively final

CountDemo.java

```java
int count = 0;
songs.forEach(s -> {
    if (s.seconds() > 200) {
        count++;
    }
});
```

- Una lambda puede usar variables locales… si nunca se reasignan: effectively final.
- La lambda puede vivir más que el método: Java copia el valor al crearla.
- Para contar: `filter(...).count()`, o un for-each común.

javac · salida recortada

```
CountDemo.java:9: error: local variables referenced from a lambda
    expression must be final or effectively final
```

Comparadores compuestos

ThreeWaysDemo.java

```java
songs.sort(Comparator.comparing(Song::artist)
        .thenComparing(Song::seconds, Comparator.reverseOrder()));
```

bash

```
Costa Verde - Garúa (220)
Costa Verde - Malecón (190)
Duna - Brisa (210)
Duna - Arena (185)
Los Andinos - Puente de piedra (245)
Los Andinos - Cumbre (190)
```

- `comparing(f)`: por la clave de `f`.
- `thenComparing(g)`: desempata.
- `reversed()` invierte todo el comparador; para un solo criterio, `thenComparing(f, reverseOrder())`.

Colecciones que reciben lambdas

CollectionLambdasDemo.java

```java
songs.removeIf(s -> s.seconds() < 200);

for (String w : text.split(" ")) {
    counts.merge(w, 1, Integer::sum);
}

for (Song s : Songs.sample()) {
    byArtist.computeIfAbsent(s.artist(), k ->
     new ArrayList<>())
            .add(s.title());
}
byArtist.forEach((artist, titles) -> IO.println
     (artist + " -> " + titles));
```

- `removeIf`: el `it.remove()` de la semana 11, sin iterador.
- `merge`: si no está, pone 1; si está, suma.
- `computeIfAbsent`: crea la lista la primera vez.
- `forEach` en un `Map`: clave y valor.

SECCIÓN 02

Sesión B: streams

02

Un bucle frente a un stream

con un bucle

```
List<String> longTitles = new
    ArrayList<>();
for (Song s : songs) {
    if (s.seconds() > 200) {
        longTitles.add(s.title());
    }
}
Collections.sort(longTitles);
```

StreamDemo.java

```
List<String> longTitles = songs.
    stream()
        .filter(s -> s.seconds() >
    200)
        .map(Song::title)
        .sorted()
        .toList();
```

Fuente, tres intermedias, una terminal.
Da `[Brisa, Garúa, Puente de piedra]`.

Cómo frente a qué: el bucle dice *cómo* (crear, recorrer, preguntar, agregar, ordenar); el stream dice *qué*.

Un stream es un pipeline

	Operaciones	Devuelve
Fuente	`list.stream()`, `Stream.of(...)`, `IntStream.range(a, b)`	un stream
Intermedia	`filter`, `map`, `mapToInt`, `sorted`, `distinct`, `limit`	otro stream
Terminal	`toList`, `collect`, `count`, `max`, `findFirst`, `anyMatch`	un resultado

- Un stream no es una colección: no guarda los elementos, no tiene índices.
- Se recorre una sola vez.

Más terminales

StreamDemo.java

```java
long dunas = songs.stream().filter(s -> s.artist().equals("Duna")).count();
int total = songs.stream().mapToInt(Song::seconds).sum();
double average = songs.stream().mapToInt(Song::seconds).average().orElse(0);
boolean anyShort = songs.stream().anyMatch(s -> s.seconds() < 180);
List<String> artists = songs.stream().map(Song::artist).distinct().sorted().toList();
```

bash

```
Duna: 2, total: 1240 s, promedio: 206.66666666666666 s
¿alguna de menos de 3 min? false; artistas: [Costa Verde, Duna, Los Andinos]
```

Lazy: nada corre hasta la terminal

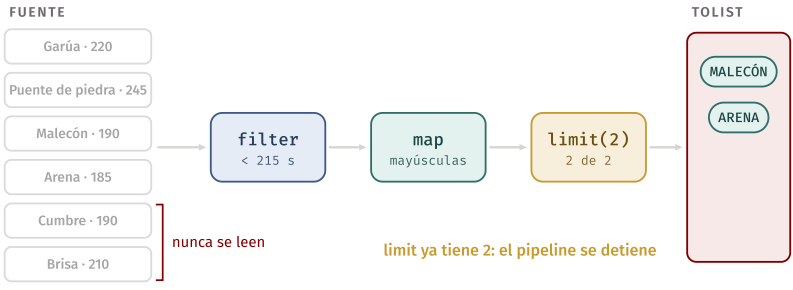
LazyDemo.java

```
1  Stream<String> pipeline = songs.stream()
2      .filter(s -> {
3          IO.println("  filter " + s.title());
4          return s.seconds() < 215;
5      })
6      .map(s -> {
7          IO.println("  map    " + s.title());
8          return s.title().toUpperCase();
9      });
10 IO.println("pipeline armado, sin operación
       terminal todavía");
11 List<String> firstTwo = pipeline.limit(2).toList
       ();
12 IO.println("resultado: " + firstTwo);
```

- Las lambdas imprimen cada vez que se ejecutan.
- ¿Qué imprime, y en qué orden?
- ¿Cuántas canciones se leen?

Los elementos pasan de uno en uno

```
songs.stream().filter(s -> s.seconds() < 215).map(...toUpperCase()).limit(2).toList()
```



LO QUE IMPRIME LAZYDEMO

```
filter Garúa                 map    Malecón
filter Puente de piedra      filter Arena
filter Malecón               map    Arena
```

Tres lecciones de `LazyDemo`

```bash
$ java -cp build LazyDemo
pipeline armado, sin operación terminal
    todavía
  filter Garúa
  filter Puente de piedra
  filter Malecón
  map    Malecón
  filter Arena
  map    Arena
resultado: [MALECÓN, ARENA]
```

- Armar el pipeline no ejecuta ninguna lambda.
- `filter` y `map` se intercalan: no hay una lista intermedia.
- `limit(2)` lleno: el pipeline se detiene. Cumbre y Brisa nunca se leen.

Un stream se recorre una sola vez

StreamDemo.java

```java
Stream<Song> once = songs.stream();
IO.println("primera vez: " + once.count());
IO.println("segunda vez: " + once.count());
```

bash

```
primera vez: 6
java.lang.IllegalStateException: stream has already been operated upon or
    closed
```

- No guarda los elementos: cuando llega a la terminal, está agotado.
- Por eso casi nunca se guarda en una variable: se crea, se encadena y se consume en la misma expresión.

Streams de primitivos y `reduce`

StreamDemo.java

```java
int total = songs.stream().mapToInt(Song::seconds).sum();
int sumOfSquares = IntStream.rangeClosed(1, 4).map(n -> n * n).sum();
Integer reduced = Stream.of(185, 210, 190).reduce(0, Integer::sum);
```

bash

```
cuadrados de 1 a 4, sumados = 30; reduce = 585
```

- `IntStream`: `int` sin boxing, con `sum`, `average` y `range`.
- `average()` da `OptionalDouble`: una lista vacía no tiene promedio.
- `reduce(0, Integer::sum)`: $((0 + 185) + 210) + 190$. Casi siempre hay una terminal más directa.

`collect`: `toList`, `joining`, `toMap`

ReportDemo.java

```java
Map<String, Integer> secondsByTitle = Songs.sample().stream()
        .collect(Collectors.toMap(Song::title, Song::seconds, Integer::sum, TreeMap::
    new));
```

bash

```
toMap: {Arena=185, Brisa=210, Cumbre=190, Garúa=220, Malecón=190, Puente de
    piedra=245}
```

- `stream.toList()`: lista no modificable. `joining(", ")`: une cadenas.
- `toMap(clave, valor)`: con claves repetidas lanza `IllegalStateException`…
- …salvo que se diga cómo combinar (`Integer::sum`).
- `TreeMap::new`: qué mapa crear. Sin él, `HashMap` y orden sin especificar.

`groupingBy`: agrupar y resumir

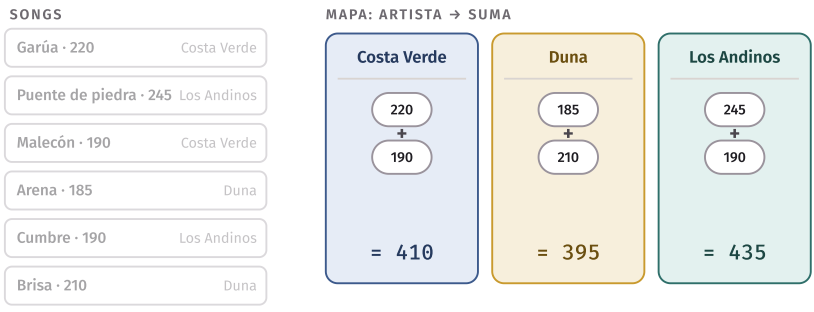
LibraryReports.java

```java
public static Map<String, Integer> totalDurationByArtist(List<Song> songs) {
    return songs.stream()
            .collect(Collectors.groupingBy(Song::artist, TreeMap::new,
                    Collectors.summingInt(Song::seconds)));
}
```

- `groupingBy(clave)`: un `Map<K, List<T>>`, un grupo por clave.
- Segundo collector: resume cada grupo. `counting()`, `summingInt(f)`, `mapping(f, toList())`.
- `TreeMap::new`: claves en orden alfabético, para imprimir y para probar.

`groupingBy`, paso a paso

```
songs.stream().collect(groupingBy(Song::artist, TreeMap::new, summingInt(Song::seconds)))
```



`{Costa Verde=410, Duna=395, Los Andinos=435}`

claves en orden alfabético porque el mapa es un TreeMap

Ejemplo: los artistas con más canciones

LibraryReports.java

```java
public static List<String> topArtists(List<Song> songs, int n) {
    return songsPerArtist(songs).entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue().reversed()
                    .thenComparing(Map.Entry.comparingByKey()))
            .limit(n)
            .map(Map.Entry::getKey)
            .toList();
}
```

bash · ReportDemo, recortada

```
top 3 artistas:   [Costa Verde, Duna, Los Andinos]
```

Costa Verde tiene 4; Duna y Los Andinos, 3: el desempate alfabético pone primero a Duna.

`Optional`: un resultado que puede no existir

LibraryReports.java

```java
public static Optional<Song> longest(List<Song> songs) {
    return songs.stream()
            .max(Comparator.comparingInt(Song::seconds));
}
```

- ¿La más larga de una lista vacía? No existe.
- El tipo de retorno avisa; quien llama decide qué hacer.
- Lo devuelven `max`, `min`, `findFirst`, `reduce` sin valor inicial.

Método	Si hay valor	Si está vacío
`orElse(x)`	el valor	`x`
`orElseThrow()`	el valor	`NoSuchElementException`
`map(f)`	`Optional` con `f(valor)`	vacío

Una caja llena o vacía



get() sobre la caja vacía lanza NoSuchElementException: mejor orElse, orElseThrow o isPresent

Error común: `get()` sin verificar

no

```
String title = songs.stream().filter(s -> s.seconds() > 600).findFirst().get().title
    ();
```

sí

```
String title = songs.stream().filter(s -> s.seconds() > 600).findFirst()
        .map(Song::title).orElse("(ninguna)");
```

- `get()` sobre una caja vacía: `NoSuchElementException`. El `NullPointerException` de siempre con otro nombre.
- Si están seguros de que hay valor, `orElseThrow()`: dice que lo pensaron.
- `Optional` es para valores de retorno, no para campos ni parámetros.

Cuándo no usar streams

Efectos secundarios dentro del pipeline

no

```
1  List<String> titles = new ArrayList<>();
2  songs.stream().filter(...).forEach(s -> titles.add(s.title()));
```

La terminal debe *producir* el resultado: `...map(Song::title).toList()`.

Un bucle que era más claro como bucle

Dos acumuladores a la vez, un `break` según un estado, comparar `a[i]` con `a[i + 1]`… La pregunta no es “¿se puede?”, sino “¿qué versión se entiende más rápido?”.

SECCIÓN 03

Laboratorio

03

Laboratorio: reportes con streams

1. Reescribir con lambdas los comparadores de la semana 11; `ComparatorsTest` debe seguir pasando.
2. Cada reporte dos veces, con bucles y con streams, con la misma prueba: total vendido, ventas por día, producto más vendido, clientes con más de *N* compras.
3. `groupingBy` ordenado; `Optional` donde no haya resultado.
4. En parejas: ¿cuál versión es más legible?

lab/Sale.java

```java
public record Sale(
        LocalDate date, String
    customer,
        String item, int quantity,
        int unitPrice) {
    public int total() {
        return quantity *
     unitPrice;
    }
}
```

Datos de ejemplo: `lab/SampleSales.java`, una cafetería en tres días.

`make test`

- `Song`, los comparadores y `ComparatorsTest`: copiados de la semana 11, sin cambios.
- `LibraryReports` y sus pruebas.
- `lab/`: `Sale`, `SampleSales` y un esqueleto de prueba.

bash

```
$ make test
...
== ReportDemo
...
[        11 tests found           ]
...
[        11 tests successful      ]
[         0 tests failed          ]
```

Salida recortada.

Resumen

- Una **lambda** es un objeto que implementa una **interfaz funcional** (un solo método abstracto); la interfaz la decide el **target type**.
- `java.util.function`: `Predicate`, `Function`, `Consumer`, `Supplier`, `UnaryOperator`, `BinaryOperator`. **Referencias a métodos**: `Song::title`.
- Variables capturadas: **effectively final**.
- `comparing`, `thenComparing`, `reversed`; `removeIf`, `merge`, `computeIfAbsent`.
- **Stream**: fuente, intermedias, terminal. **Lazy**, de uno en uno, una sola vez.
- `collect` con `groupingBy`, `counting`, `summingInt`, `toMap`; `TreeMap::new` para un orden fijo.
- `Optional`: `orElse`, `orElseThrow`, `map`; nunca `get()` a ciegas.

Para la próxima clase

- Lecturas de la semana 13: Eck, sec. 4.1; R. C. Martin, *The Single Responsibility Principle* y *The Open Closed Principle*.
- **P3** se entrega esta semana. **P4** (Reportes) se publica: los reportes se escriben con lo de hoy.
- Terminar el laboratorio: cada reporte con bucles y con streams.

Pregunta para pensar

Ya pueden pasar comportamiento como argumento: un comparador, un predicado, una regla de descuento. ¿Qué parte de su proyecto cambiaría si el código que vende recibiera la regla de precio en lugar de decidirla con un `if`?